Distributed Systems: Paradigms and Models

Game of Life - Project Report

Stefano Forti (481183)

February 21, 2016

Table of Contents

1 Introduction

The aim of this report is to describe the project *Game of Life* (GoL) and its implementation, also reporting experimental results to validate its design and performance. The project is written in Java 8 and consists of four main components:

- a **sequential** version that represents the foundation for
- a multi-threaded version, implemented by means of the **Java Thread** mechanisms,
- a multi-threaded version, implemented within the **Skandium** parallel skeletons framework and
- a multi-threaded version, implemented within the **Muskel2** framework.

In what follows those four solutions are described. Particular attention is given to the parallelisation process which precedes the concurrent implementation phase. Subsequently, the results obtained whilst evaluating *service time*, *scalability*, *speedup* and *efficiency* of each version are discussed. All tests were run on an Intel Xeon E5-2650 (2.00GHz) processor with 8 cores and 16 contexts. To conclude the report, a brief user guide to launch the program and some tests is given.

2 The Sequential GoL

The sequential version of the program is in the `edu.spm.stefano.gameoflife` package. It includes three classes – `Board`, `GraphicBoard` and `Interval` – that are the exploited in all the parallel implementations and a main class, `GameOfLife`. Hereinafter, all those classes are briefly described, highlighting the major design choices.

2.1 The `Interval` class

This simple class provides a way of specifying row-wise chunks in our GoL boards. Instances of this class have two fields: one indicating a starting row for the chunk and the other counting the number of rows to be included.

2.2 The `Board` class

The `Board` class implements the sequential algorithm for GoL and it is of crucial importance for the entire project. Particularly, a `Board` object B stores the dimensions (m, n) of the GoL board and two `bytes` matrices with those dimensions: `boardFrom` and `boardTo`. At the end of each step of the algorithm, the following invariant holds: `boardFrom` is B^k and `boardTo` is B^{k+1}, with the superscript indicating the generation of the board B. Before computing a further generation, the two boards must be swapped by calling the provided method `swapBoards()`.

The core of the algorithm is the `countAliveNeighbours(int i, int j)` that returns the number of alive cells adjacent to $B_{i,j}$. In fact, it revealed to be a source of many possible optimisations. Since all cells must have eight neighbours, the ones on the border rows and columns have some neighbours on the other side of the board, as shown through an example in Figure 1. A trivial

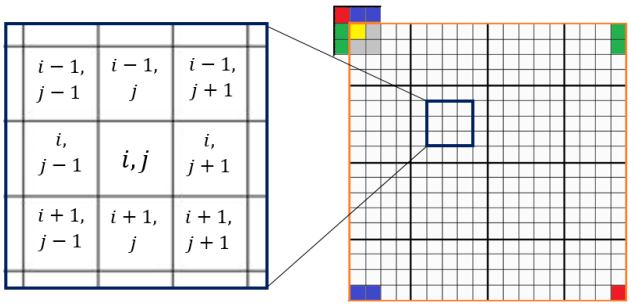

Figure 1: On the left handside, the neighbours of a cell $B_{i,j}$ that does not lie on the border of the board. On the right handside the neighbours of the left topmost cell $B_{0,0}$ (in yellow) are coloured: not only the grey ones are considered adjacent, but also the red, green and blue ones.

solution to this problem would make use of the positive modulo operator, applying it to the indexes of the adjacent cells (e.g. `((i + 1) % m + m) % m)`). Unfortunately, this approach leads to huge inefficiencies, since the actual implementation of the modulo operation on the current architectures exploits some multiplication operations, thus taking a considerable amount of clock cycles to execute [6]. This is the reason why, as shown in Listing 1, the proposed implementation substitutes the modulo with a simple check for border cells, what makes the code almost four times faster to run in our settings.

```
1   private int countAliveNeighbours(int i, int j) {
2       int result = 0;
3       int prevRow = i - 1, succRow = i + 1,
4       prevColumn = j - 1, succColumn = j + 1;
5
6       if (i == 0)
7           prevRow = M - 1;
8       if (j == 0)
9           prevColumn = N - 1;
10      if (j == N - 1)
11          succColumn = 0;
12      if (i == M - 1)
13          succRow = 0;
14
15      result = boardFrom[prevRow][prevColumn]
16              + boardFrom[prevRow][j]
17              + boardFrom[prevRow][succColumn]
18              + boardFrom[i][prevColumn]
19              + boardFrom[i][succColumn]
20              + boardFrom[succRow][prevColumn]
21              + boardFrom[succRow][j]
22              + boardFrom[succRow][succColumn];
23
24      return result;
25  }
```

Listing 1: A function that counts the alive neighbours of a cell without using any modulo operation.

The `countAliveNeighbours(int i, int j)` method is called by the `makeStep(int startRow, int nRows)` method which, in turn, performs the four checks reported in Section 2 by means of a `switch` that compiles into more efficient bytecode with respect to an `if`.

Furthermore, the `Board` class provides three methods to initialise `fromBoard` to some B^0 at the

beginning of a game, choosing among:

1. a random B^0, with `initializeBoard()`,
2. a pseudo-random B^0, useful for obtaining always the same matrix during a test, with `initializeBoard(long seed)`,
3. a glider configuration, actually used to validate the implementation.

Eventually, the `Board` method `splitBoard(int NTHREADS)` returns an array of `Intervals` that divide the matrix row-wise into `NTHREAD` balanced parts. This last method is fundamental for the parallel implementations and the row-wise partitioning will be discussed further later on.

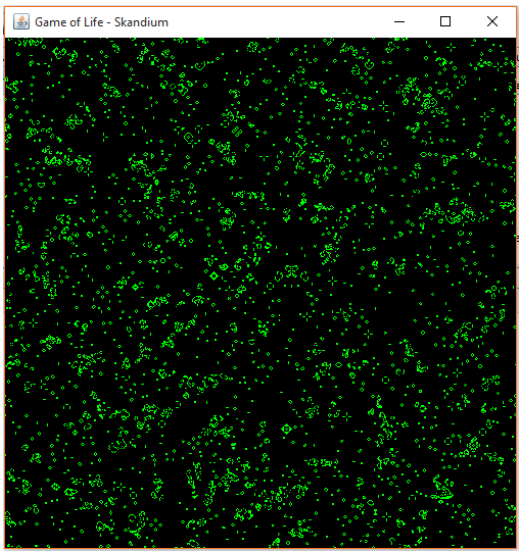

Figure 2: An example of the graphical version of GoL.

2.3 The `GraphicBoard` class

This class, extending the Swing `JPanel` in the `javax.swing` package, is used to graphically render any `Board` instance. Despite the fact that Swing is not thread safe [1], this class works pretty fine, having the method `updateScreen(Graphics g)` repainting the board whenever another complete image which was previously drawn is available to be drawn again (`ImageObserver.FRAMEBITS` set). An example of such rendering, in which each pixel represents a cell of the board is shown in Figure 2.

3 Parallelising GoL

This section describes the design phase of the parallel versions of GoL. Both the adopted data parallel pattern and its cost model are hereinafter reported.

3.1 The Map Pattern

The natural pattern to parallelise GoL is definetely a map, which can be described as

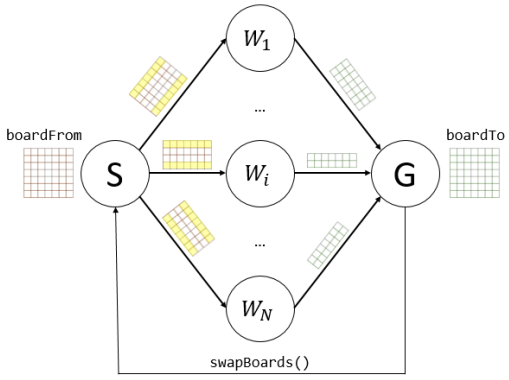

Figure 3: A graphical sketch of the data parallel pattern adopted for GoL. The two yellow rows are needed to process the chunk borders and are the ones before and after the chunk respectively.

$$Map(B.\texttt{makeStep(startRow, nRows)}) = \overleftarrow{(\triangleleft_{scatter} \bullet [|((B.\texttt{makeStep(startRow, nRows)}))|]_N \bullet \triangleright_{gatherAll}((B.\texttt{swapBoards()})))_c}$$

by means of the RISC-pb^2 formalism [5] and works logically as depicted in Figure 3. The map is repeated on the output data until condition $c = \neg(G^{th}generation)$ holds, that is G times to reach B^G from B^0. It is worth noting that the N workers should also receive from the $scatter$ two extra rows in order to compute the next state, i.e. the row immediately before the assigned sub-matrix and the one immediately after.

In all our concurrent versions, the matrix is logically split row-wise by giving to each thread a chunk of the board to process. This can be safely done thanks to the implementation of `Board` that ensures that `boardFrom` is read-only and that each thread writes only onto its portion of `boardTo`, invoking the method B`.makeStep(int startRow, int nRows)` on its `Interval`. As a consequence, useless copies of the board are avoided and so is any unnecessary communication overhead; in a sense, the Bernstein conditions and the *owner computes rule* [8] are satisfied in this scenario hence it is not needed to actually split the board. Finally, the $gatherAll$ represents barrier at the end of each step of the computation.

Before studying the cost model of this approach, a note on the chosen partitioning of the data is due. Up to our knowledge, Java does not support two-dimensional arrays but stores matrices as arrays of arrays; hence, there is no guarantee that all the elements of a board will be stored contiguosly, either in row-major or column-major order. Anyway, each row will be stored continguosly in memory thus making the row-wise split way better than both the column-wise and the sub-matrices partitioning with respect to cache reuse.

3.2 The Cost Model

According to what stated before, it is possible to derive a convenient cost model to later interpret the empirical results. For the Map pattern we have that the completion time for computing one generation is

$$T_s = T_{scatter} + T_F + T_{gatherAll} + T_{swap}$$

where $F = [|((B.\texttt{makeStep(startRow, nRows)}))|]_N$ and $swap =$ `swapBoards()`. Assuming no communication overhead, for what stated in Section 3.1, we can rewrite the latter as

$$T_s \simeq T_F + T_{swap} \simeq T_F$$

where the last step is justified considering the swap of two pointers and the barrier awaiting time negligible, during relatively coarse-grained computations. Therefore, the completion time can be approximated as

$$T_c = G \times T_s \simeq G \times T_F$$

with $T_F \simeq \frac{T_{seq}}{N}$ ideally and G being the number of board generations to be computed. Eventually, some performance degradation is expected, due to threads management and JVM overhead.

3.3 Java Threads

The first concurrent version of the program which exploits the standard Java `Thread` mechanisms is in the `edu.spm.stefano.gameoflifemultithreaded` package. It is composed of two classes: the `Consumer`, implementing the `Runnable` interface and a main class `GameOfLifeMultiThreaded`.

```
1         final CyclicBarrier barrier =
2               new CyclicBarrier(NTHREADS, board::swapBoards);
3         ExecutorService threadPool = Executors.newFixedThreadPool(NTHREADS);
4         final long startTime = System.currentTimeMillis();
5         for (int j = 0; j < NTHREADS; j++){
6             threadPool.execute(new Consumer(
7                   board, bounds[j].a, bounds[j].b, steps, barrier));
8         }
9         threadPool.shutdown();
10        threadPool.awaitTermination(10, TimeUnit.MINUTES);
```

Listing 2: The core of the main class for the Java `Thread` version of GoL.

3.3.1 The `GameOfLifeMultiThreaded` class

The main class of this package is utterly important since it manages the thread pool and defines the `CyclicBarrier`, as listed in Listing 2. The presence in Java of `CyclicBarriers` simplified dramatically the implementation phase: this data structure supports an optional `Runnable` command that is run once per barrier point, after the last thread in the pool arrives, but before any threads are released [1]. Therefore, it is up to the barrier to sequentially execute the `swapBoards()` method at the end of each iteration, updating the shared state.

3.3.2 The `Consumer` class

Each instance of `Consumer` receives a pointer to the current `Board` object, the chunk of the matrix – specified as a starting row `start` and a number of rows to compute `n` –, a pointer to the `CyclicBarrier` defined in the main and the number of generations to be computed (`steps`). The code for the overridden `run()` method is shown in Listing 3.

```
1     public void run() {
2         for (int i = 0; i < steps; i++) {
3             board.makeStep(start, n);
4             barrier.await();
5       }
6     }
```

Listing 3: The overridden `run()` method for the `Consumer` class (`throw` clauses are not shown).

3.4 Skandium

Skandium[4][3] is a Java based algorithmic skeleton library for high-level parallel programming of multicore architectures. The Skandium implementation of GoL is in the `edu.spm.stefano.gameoflife-skandium` package and comprehends a main class and the classes `SplitBoard`, `ComputeSteps` and `MergeResults` that represent the $scatter$, the $computation$ and the $gatherAll$ phases respectively.

```
1         Skandium skandium = new Skandium(NTHREADS);
2         Skeleton<Board, Board> forLoop =
3                 new For(
4                     new Map<>(
5                     new SplitBoard(NTHREADS),
6                     new ComputeSteps(board),
7                     new MergeResults(board)),
8                 steps);
9         Stream<Board, Board> stream = skandium.newStream(forLoop);
10        Future<Board> future = stream.input(board);
11        Board res = future.get();
```

Listing 4: The core of the `GameOfLifeSkandium` main class.

3.4.1 The `GameOfLifeSkandium` class

The main class, whose core is in Listing 4, and particularly lines 3-8, can be read beside the RISC-pb^2 in Section 3.1 where the feedback arrow has been implemented as a Skandium `Muscle For`. This made the Skandium implementation almost straightforward, after the design step.

3.4.2 The `SplitBoard` class

This class implements the `Split<Board, Interval>` interface required by the Skandium `Map` skeleton in this scenario. Basically, it performs the data partitioning of the board before each compute phase, whereas the implementation described in Section 3.3 computes the matrix chunks once for all before starting the computation. The overridden `split(Board b)` simply wraps the B.`splitBoard(NTHREADS)` call and obtains an array of `Intervals` to be streamed to the `ComputeSteps` object.

3.4.3 The `ComputeSteps` class

This class implements the `Execute<Interval, Board>` interface needed by Skandium in GoL. It matter-of-factly wraps the B.`makeStep(startRow, nRows)` method invocation. This is the code

executed by each worker in parallel at each step; the output goes to the `MergeResults` object.

3.4.4 The `MergeResults` class

This class the required `Merge<Board, Board>` Skandium interface. As the classes describe above, it only wraps the B.`swapBoard()` sequential execution.

3.5 Muskel2

Muskel2 [7][2] is a Java 8 library for parallel and distributed computing that implements the Reactive Streams specifications. It is based on the recently added Java `Streams`. The Muskel 2 implementation consists only of a main class; its core is shown in Listing 5 and commented below.

```
1         MuskelContext context = MuskelContext.builder().local()
2                         .defaultPoolSize(NTHREADS).build();
3         Interval[] bounds = board.splitBoard(NTHREADS);
4
5         for (int k = 0; k < steps; k++) {
6             MuskelProcessor.from(bounds)
7                     .withContext(context)
8                     .map((Interval b) -> {
9                         board.makeStep(b.a, b.b);
10                        return b;
11                    }, local())
12                    .toBlocking()
13                    .first();
14            board.swapBoards();
15        }
```

Listing 5: The core of the `GameOfLifeMuskel2` main class.

After instantiating a `LocalMuskelContext` that manages the needed thread pool and resources, a stream of `Intervals` is the input to a map, performing the required computations. The `toBlocking()` call acts as a barrier and the board swapping is sequentially executed by the main.

4 Performance Analysis

In order to test the performance, the completion time of each solution was measured by invoking the Java `System.currentTimeMillis()`, directly within the code; all GoL executions print their T_c. Two simple bash scripts `measure.sh` and `measureAll.sh` were used to run 10 times

- the sequential algorithm with $G = 500$,
- each parallel implementation, varying the number of workers from 1 to 16, with $G = 500$

averaging the resulting completion time.

These tests were run over different sizes of the input board: 4000×4000, 2000×2000 1000×1000, 500×500, 250×250. For all the tests B^0 was initialised with the seed 481183. The results – plotted

in Figure 5,6, 7, 8, 9 – seem to confirm the observations of Section 3.2: fine grained (500×500, 250×250) computations suffer more from the overhead due to threads management and sequential operations, leading to worst scalability, speedup and efficiency as the number of threads increases.

5 Conclusions and Future Work

To sum up, this project shows two Java libraries for parallel skeletons at work on a simple, yet interesting, example. Unfortunately, due to the high number of users working on the available Intel Xeon E5-2650 (2.00GHz) processor, it has not been possible to establish which between Skandium and Muskel2 performs better. Surely, Muskel2 wins in terms of code succinctness, after getting familiar with the new Java 8 syntax, whereas Skandium is easier to learn but requires to define many classes even for this simple application. On the other hand, the Java `Thread` implementation turned out to be, on average, the one with the best performances, as expected.

Future work would include to repeat some tests in the ideal conditions and in such a way they are statistically significant, i.e. without other users on the processor and repeating each test at least 30 times. Finally, it would be interesting to add some known patterns to initialise B^0, other than the glider.

6 A Brief User Guide

6.1 Create the jar file

The project comes with a Maven `POM.xml` file; after installing Maven, to create the executable jar file, simply launch the script `install.sh` which also resolves the dependencies with Skandium and Muskel2 by adding them to the local Maven repository. The file `GameOfLife-1.0-jar-with-dependencies.jar` in the `target` folder is the executable.

6.2 Launching the jar

The jar file can be launched from the project folder issuing the command

```
java -jar ./target/GameOfLife-1.0-jar-with-dependencies.jar
```

with the options shown below in Figure 4, implemented through the Apache Commons CLI library.

```
usage: GameOfLife [-f <arg>] [-g] [-glider] [-m <arg>] [-n <arg>] [-N <arg>] [-s <arg>] [-t <arg>]
 -f <arg>   select the GoL version to be run:
            seq (sequential), mt (Java threads),
            mu2 (Muskel2), sk (Skandium)
 -g         activate the graphics visualisation
 -glider    use the "glider" initialisation
 -m <arg>   number of rows for the GoL board
 -n <arg>   number of columns for the GoL board
 -N <arg>   number of threads for parallel implementations
 -s <arg>   initialise the matrix by means of a seed
 -t <arg>   number of generations for GoL
```

Figure 4: The available options in GoL's CLI.

Figure 5: Average completion time, speed up, scalability and efficiency with $G = 500$ and a board 4000×4000. The ideal curve (or part of it for scalability and speedup) is also plotted.

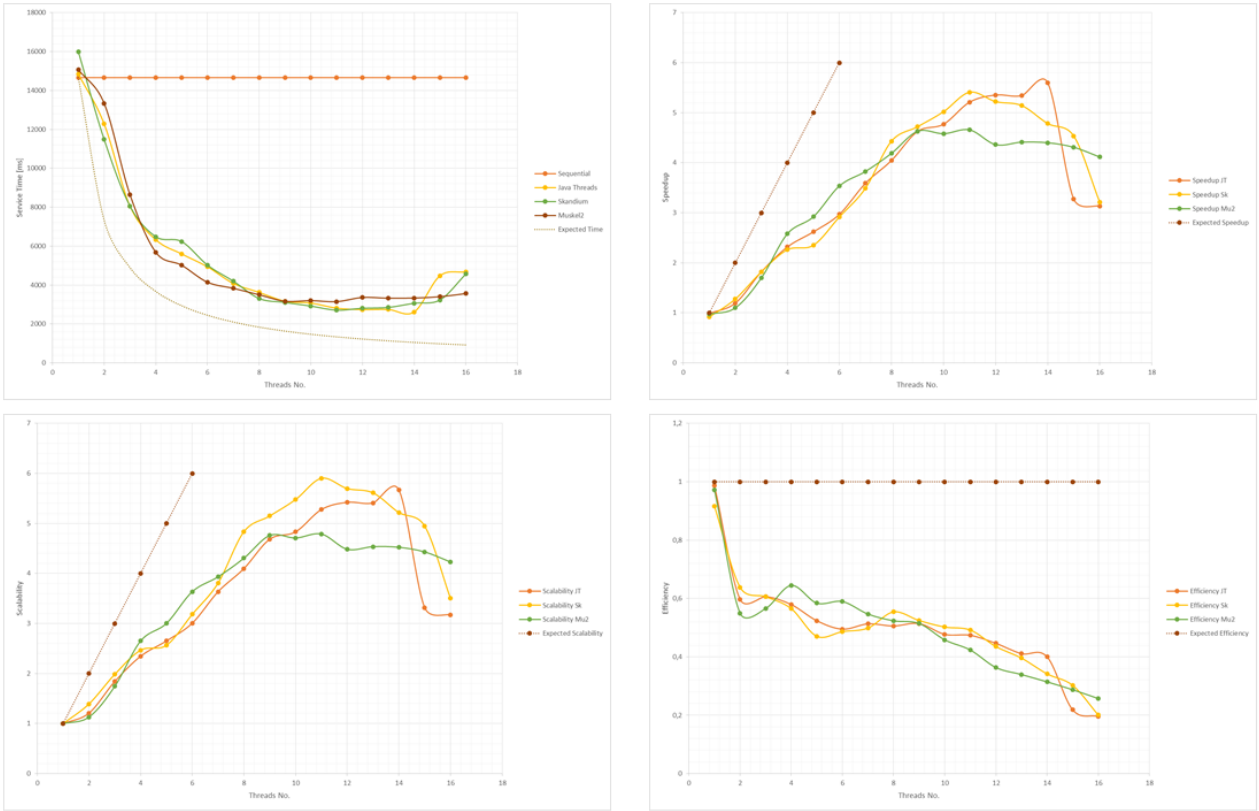

Figure 6: Average completion time, speed up, scalability and efficiency with $G = 500$ and a board 2000×2000.The ideal curve (or part of it for scalability and speedup) is also plotted.

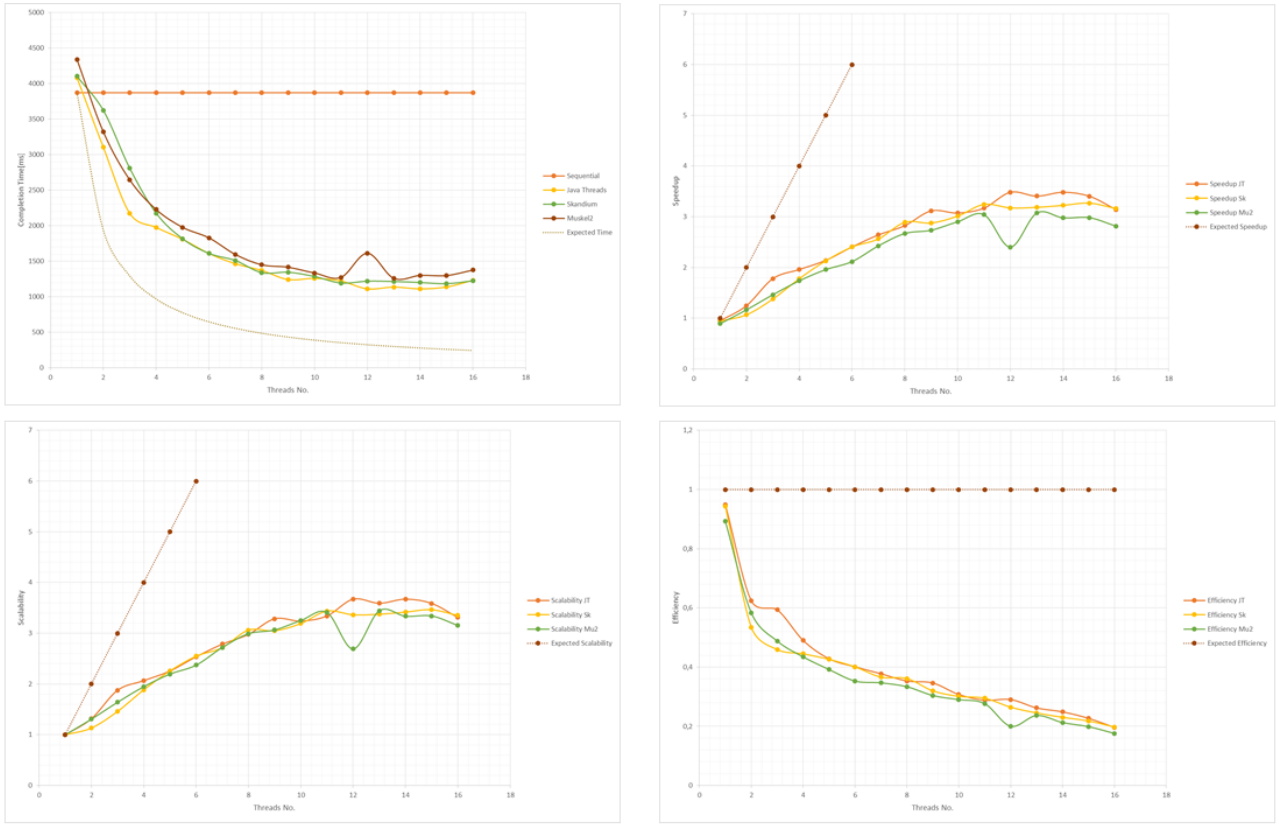

Figure 7: Average completion time, speed up, scalability and efficiency with $G = 500$ and a board 1000×1000. The ideal curve (or part of it for scalability and speedup) is also plotted.

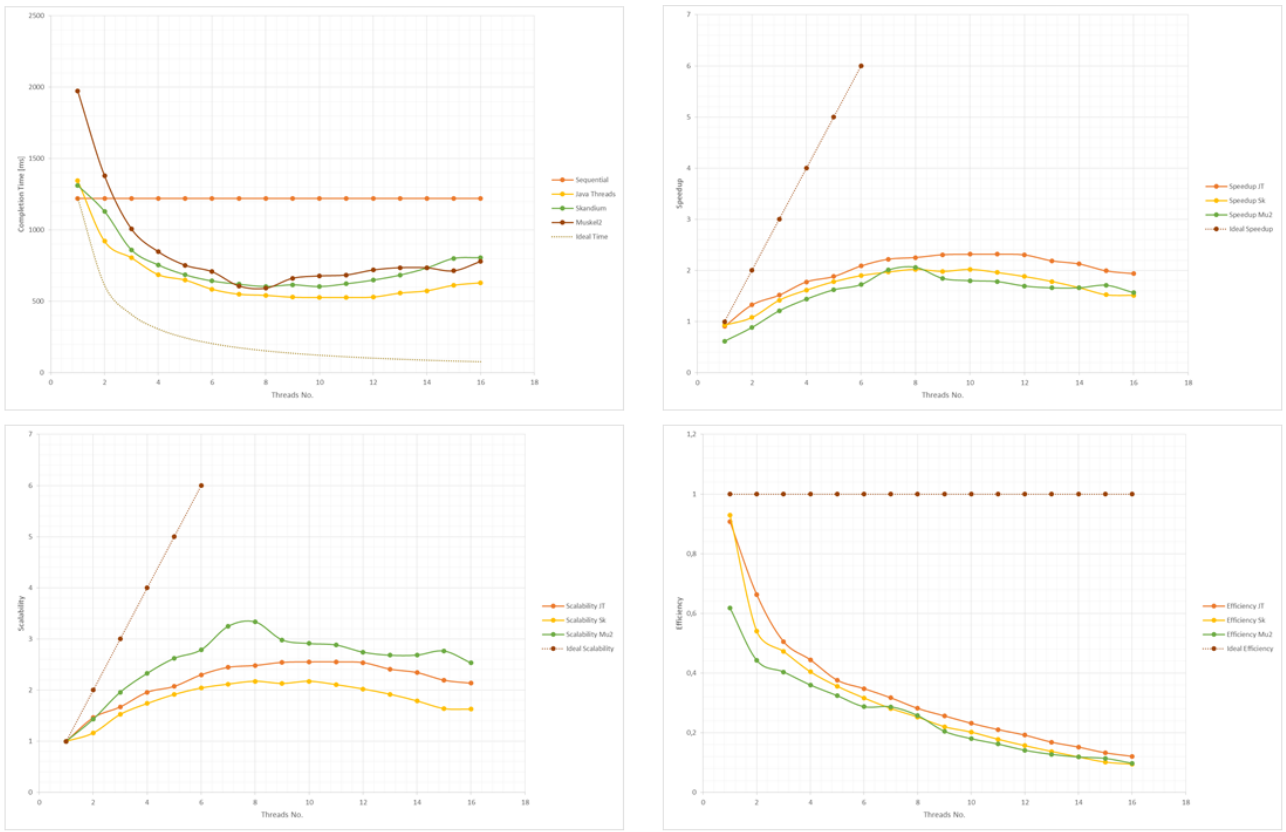

Figure 8: Average completion time, speed up, scalability and efficiency with $G = 500$ and a board 500×500. The ideal curve (or part of it for scalability and speedup) is also plotted.

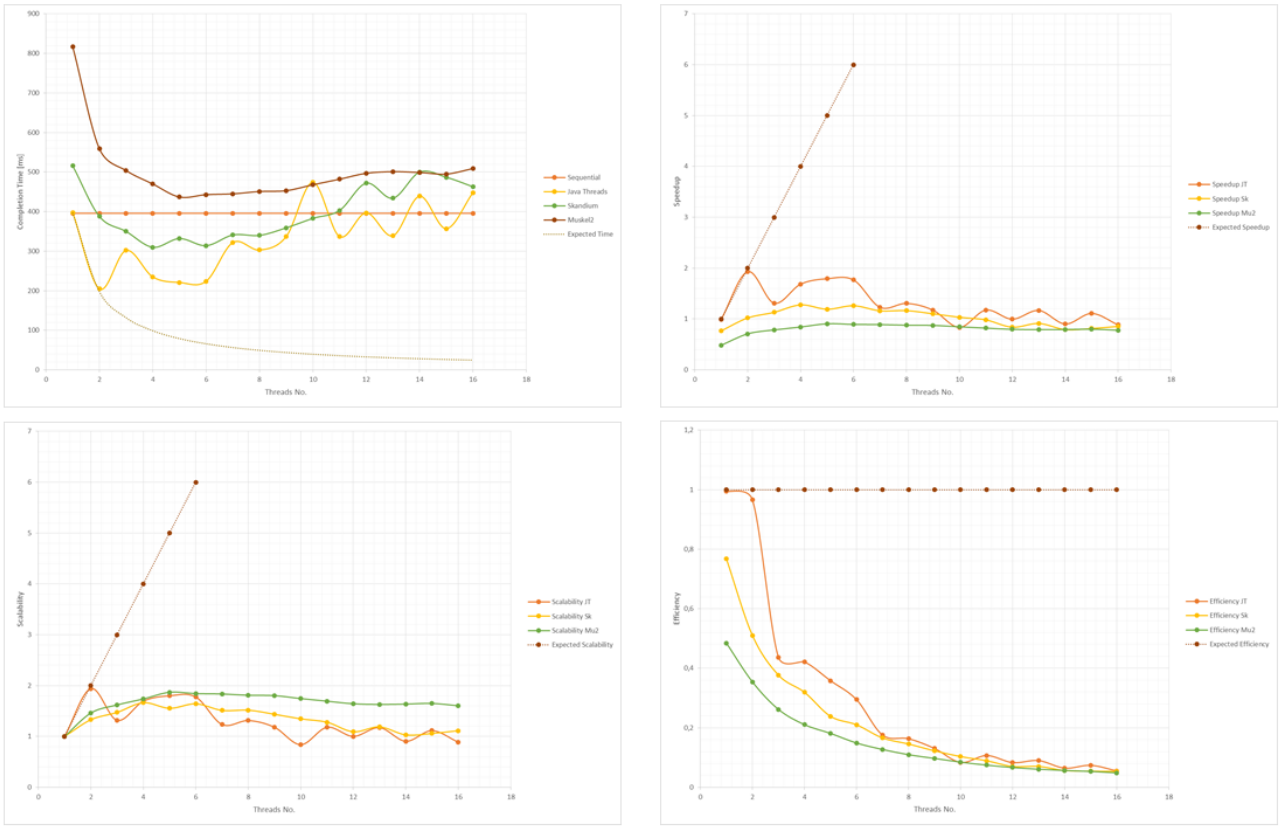

Figure 9: Average completion time, speed up, scalability and efficiency with $G = 500$ and a board 250×250. The ideal curve (or part of it for scalability and speedup) is also plotted.

References

[1] Java Platform Standard Edition 8 Documentation. https://docs.oracle.com/javase/8/docs. Accessed: 20/02/2016.

[2] Muskel2 on Github. https://github.com/chrysimo/muskel2. Accessed: 20/02/2016.

[3] Skandium on Github. https://github.com/mleyton/Skandium. Accessed: 20/02/2016.

[4] Skandium website. http://backus.di.unipi.it/m̃arcod/SkandiumClone/skandium.niclabs.cl/index.html. Accessed: 20/02/2016.

[5] Danelutto, M. *Distributed systems: paradigms and models*. Teaching notes, 2013.

[6] Money, D., and Harris, S. L. *Digital design and computer architecture*. Morgan Kaufmann Publishers, 2007.

[7] Simonelli, C. *Programmazione parallela strutturata in Java8: Muskel2*. MSc Thesis, 2015.

[8] Vanneschi, M. *High performance computing: parallel processing models and architectures*. Pisa University Press, 2014.